UNPUBLISHED PRELIMINARY DATA

[redacted]

UNIVERSITY OF MARYLAND 1807-1856-1920

UNIVERSITY OF MARYLAND COMPUTER SCIENCE CENTER

COLLEGE PARK, MARYLAND

Technical Report TR-64-9
NsG-398 June 1964

Utilizing the Macro Generator of IBMAP for the IBM 7090/7094*

by

Gerald M. Berns

IBM Corporation

*The computer time for this project was supported by the National Aeronautics and Space Administration Grant NsG-398. The author performed the work while assigned to the Computer Science Center as the IBM Systems Representative.

ABSTRACT

This report is designed to be a manual for using the macro capabilities of the IBMAP language. The concept of macro instructions is described and instructions for the definition and use of ordinary macros, nested macros, and recursive macros are given, with many illustrative examples. Special attention is given to the concept of "set-value" and to the SET, IFT, IFF, and IRP pseudo-operations.

Also included are several Fortran-like macro definitions which might be of value to the IBMAP programmer.

TABLE OF CONTENTS

UTILIZING THE MACRO GENERATOR OF IBMAP FOR THE IBM 7090/7094

INTRODUCTION

The IBMAP macro generator has more capability than had the macro generators of the assembly languages which preceded it. Unfortunately, applications programmers in the past have made little use of the power available in the macro generators of the older languages, and they show little sign of taking advantage of the greater macro power available to them now. Much of the programmers' reluctance to utilize macros can be attributed to the "mystique" which seems to surround these pseudo-operations--a "mystique" that arises in the very name "macro" itself and that is perpetuated by the scarcity of information on the subject; I am aware of no IBM Education Center classes that teach the use of macros in any depth. And if programmers speak in hushed tones of macro instructions, they shudder at the sound of "nested" macros and positively blanch whenever a "recursive" macro happens to be mentioned. Set-values, the SET pseudo-operations, and coupled "if" statements (and uncoupled, too for that matter), are practically unheard of, and the IRP operation has on it the dust of years of disuse.

What is required is a straight-forward presentation of what a macro instruction is, how one is defined to do a particular job, and how the defined macro is used. Without this information a large part of the improvement in IBMAP over its predecessors is lost. This point is made strongly and simply by considering the acronym "IBMAP" itself; it stands for the IBM <u>Macro</u> Assembly Program.

MACRO INSTRUCTIONS

Macro instructions have two major uses: they save the programmer the time and effort involved in writing repetitious blocks of code, and they enable the programmer to accomplish tasks within an assembly language program that cannot be done in any other way.

A macro instruction, once it is defined, is used ("called") by writing the name of the macro instruction in the operation field (beginning in column 8) just as any hardware or pseudo operation is coded; all that is required is that each macro instruction be defined before its initial use in a program deck. Each macro instruction used is expanded by the assembly program at the place in the program deck at which it occurs. Each macro instruction is thus automatically replaced by the pertinent instructions from the macro definition. In this way the programmer is relieved of the chore of writing the blocks of code, and in this way can code be generated in the program deck which can be generated by no other method.

The Basics of Defining and Using a Macro Instruction

The MACRO and ENDM Pseudo Operations

Suppose in a program deck there existed the following code:

```
CLA     XA
ADD     XB
STO     XC
 .
 .
 .
CLA     X
ADD     Y
STO     Z
 .
 .
 .
CLA     XB
ADD     Z
STO     A
 .
 .
 .
```

The programmer could have saved himself the effort of writing the repetitious sequences of instructions by defining a simple macro instruction to do the same job, such as:

```
columns          1           8           14-16
                 STOSUM      MACRO       A, B, C
                             CLA         A
                             ADD         B
                             STO         C
                             ENDM        STOSUM
```

where the symbol used in the location field of the MACRO pseudo-operation becomes the name of the macro instruction (it may be the same as any other valid symbol used in the program). The variable field of the MACRO pseudo-op contains a list of "dummy" arguments (any of which may be the same as any symbol used in the program or any other macro definition), each of which is replaced by real arguments when the macro instruction is used in the program deck. Each macro definition must be ended by the pseudo-op ENDM with the name of the defined macro operation in the first variable field (or else blanks).

The programmer, having properly defined his macro operation, may now use it in his program deck as follows:

```
8
STOSUM      XA, XB, XC
   .
   .
   .
STOSUM      X, Y, Z
   .
   .
   .
STOSUM      XB, Z, A
```

The instructions actually assembled in the program deck will be identical to the instructions that were written earlier without using the macro instruction, but the programmer, by using the macro instruction STOSUM, has saved the time required to write the repetitious code. The time saving in many cases can be substantial.

A macro definition is limited to 63 substitutable ("dummy") arguments, each one of which must not be more than 6 characters long. These arguments may be used to represent the location field, the operation field, the variable field (as has already been shown), and the comments field of any instruction in the macro definition-- or any one, two, or three of these fields. For example, consider the macro definition

```
QPOLY     MACRO     COEFF, LOOP, DEG, T, OP
          AXT       DEG, T
          LDQ       COEFF
LOOP      FMP       GAMMA
          OP        COEFF + DEG + 1, T
          XCA
          TIX       LOOP, T, 1
          ENDM      QPOLY
```

Suppose it were used in a program deck as follows:

```
          CLA       X
           .
           .
           .
          STO       Y
X015      QPOLY     C1 - 4, FIRST, 5, 4, FAD
```

then the code that would be generated by the macro QPOLY is:

```
X015      AXT       5, 4
          LDQ       C1 - 4
FIRST     FMP       GAMMA
          FAD       C1 + 2, 4
          XCA
          TIX       FIRST, 4, 1
```

Note that the location field of QPOLY contains the symbol X015 and that this symbol becomes the name of the first instruction in the macro expansion. Note also that the symbol GAMMA is not a dummy argument since it does not appear in the variable field of the MACRO pseudo-op; it is an ordinary symbol and is called "text".

A macro definition, although it takes space on the coding sheet to write it, does not take core space away from the executable program. When the assembly program encounters a macro definition it inserts it in a special form into the "macro skeleton table" (space in the assembler set aside for macro definitions) and places the name of the macro instruction in the operation code dictionary. Thus, the macro definition requires core only during assembly - not during execution.

If a macro instruction name is the same as the name of any hardware or pseudo-operation (such as, for example, CLA) then the operation is redefined by the macro definition.

A macro definition may contain in it any hardware instructions, pseudo-operations, macro instructions, or macro definitions. If it contains another macro instruction, then the macro instruction within is called a "nested" macro.

"Dummy" arguments in the variable field of a MACRO pseudo op may be separated one from the other by any of =+-*/,'(). The following are all equivalent and valid:

```
XYZ          MACRO        LOAN, RATE, INTRST

XYZ          MACRO        LOAN*RATE=INTRST

XYZ          MACRO        LOAN (RATE)INTRST
```

Parentheses, when used, must be used in pairs. Only commas or parentheses may be used to separate the arguments of a macro <u>instruction</u>. When in doubt about which delimiter to use, use the comma; it is always valid.

The apostrophe is used to insert a substitutable argument into any field. For example, look at the definition of the macro instruction MESS:

```
MESS    MACRO       A, B, C
        BCI         A,'B' ERROR. CONDITION'C' IGNORED
        ENDM        MESS
```

If used as follows:

```
        MESS        6,WRITE, S
```

it produces:

```
        BCI         6,WRITE ERROR.  CONDITIONS IGNORED.
```

If it is used as:

```
        MESS        6,FIELD,
```

it produces:

```
        BCI        6,FIELD ERROR.  CONDITION IGNORED.
```

It may also be used as follows:

```
        MESS       8,(READ REDUNDANCY),
```

which produces:

```
        BCI        8,READ REDUNDANCY ERROR. CONDITION IGNORED.
```

Observe that a trailing comma indicates that the pertinent substitutable argument is to be replaced by nothing (a "null" field), and that a comma followed by an open paren does not indicate a null field - in this case the comma is redundant but allowable. Also note that everything (blanks included) within a pair of parentheses replaces the appropriate "dummy" argument; this is in fact the only way to have an imbedded blank in the variable field of a macro instruction without stopping the variable field scan. Any macro instruction argument may be placed within parentheses if desired - this feature is not restricted to only those arguments containing imbedded blanks.

Another example of a macro definition:

```
MN      MACRO      A, B, C
        A  B
C
        ENDM       MN
```

when used as follows:

```
        MN         (AXT   10,1)(BEGIN ROUTINE)(ALPHA TRA  BETA)
```

It produces:

```
        AXT    10,1 BEGIN ROUTINE
ALPHA   TRA    BETA
```

Macro Related Pseudo-Operations and Concepts

To appreciate the possibilities inherent in IBMAP macro definitions, it is necessary at the onset that the reader become familiar with the workings of the IRP, SET, IFT and IFF pseudo-operations and understand the set- or S-value concept.

IRP - the Indefinite Repeat Pseudo-Operation

Suppose one found that he had occasion to write several times in a program deck code like the following:

```
CLA     CHAS
FAD     XY
FAD     Z
FAD     =012
FAD     AL
FAD     RUTH
 .       .
 .       .
 .       .
```

but that, on each occurrence of this particular block of code, a different number of FAD's were required. Using IRP in the macro definition this problem is easily solved, as follows:

```
SUM     MACRO   A,B
        CLA     A
        IRP     B
        FAD     B
        IRP
        ENDM    SUM
```

The macro instruction is used as follows:

```
SUM     CHAS(XY,Z,=012,AL,RUTH)
```

The code generated is identical to that shown above. Using SUM with fewer (or more) subarguments works equally well:

```
SUM     A(B,C)  produces    CLA  A
                            FAD  B
                            FAD  C
```

```
SUM        A(B)                         CLA   A
      or            produces            FAD   B
SUM        A,B
```

In a macro definition an IRP occurring with one "dummy" argument in the variable field denotes the beginning of an "IRP loop", and an IRP with a blank variable field denotes the end of an "IRP loop". Within this loop each argument of the macro instruction within the parentheses (each is called a "subargument") replaces the pertinent "dummy" argument each time through the loop, and the loop is negotiated as many times as there are subarguments. If the pertinent argument is null - that is, if there are no subarguments, the entire "IRP loop" is eliminated.

Suppose one had to code the polynomial $AX^4 + BX^3 + CX^2 + DX + E$, which can be rewritten $(((AX + B)X + C)X + D)X + E$, where A,B,C and D are called "coefficients" and E is a constant which may be considered to be the coefficient of $X^0(1)$. The code might be:

```
CLA    A
XCA
FMP    X
FAD    B
XCA
FMP    X
FAD    C
XCA
FMP    X
FAD    D
XCA
FMP    X
FAD    E
```

Note that the repeating group of instructions is

```
XCA
FMP    X
FAD    COEFF     (except the first, or high order
                  coefficient)
```

A macro might be defined for just this repeating group:

```
REPEAT      MACRO     COEFF,VAR
            IRP       COEFF
            XCA
            FMP       VAR
            FAD       COEFF
            IRP
            ENDM      REPEAT
```

Another macro, POLY, might be defined to solve the polynomial, using the nested macro instruction REPEAT:

```
POLY        MACRO     COEFF1,COEFF,VAR
            CLA       COEFF1
            REPEAT    (COEFF)VAR
            ENDM      POLY
```

POLY, if used as follows:

```
POLY        A(B,C,D,E)X
```

produces the same code given above. COEFF is written within parentheses in the variable field of the nested macro instruction REPEAT because it will be replaced by subarguments and the transmission of all subarguments is desired.

POLY might have been defined in one macro definition, eliminating the nested macro, as follows:

```
POLY        MACRO     COEFF1,COEFF,VAR
            CLA       COEFF1
            IRP       COEFF
            XCA
            FMP       VAR
            FAD       COEFF
            IRP
            ENDM      POLY
```

If used as above it produces the same code.

Note that the macro POLY, once defined as above, may be used to evaluate a polynomial of any order, not just one of fourth order!

The pseudo-op IRP may only be used in a macro definition; it is undefined elsewhere. Nested "IRP loops" (i.e. "IRP loops" within "IRP loops") are not allowed; however, nested macro instructions which themselves contain "IRP loops" are allowed within an "IRP loop" and this type of coding may continue to any "depth" (level of nesting of macros).

Set-Value

A set- or S-value is the "immediate" value that a symbol is assigned during the first pass of the assembler, and it is the first pass of the assembler that is of importance in the writing of macros. During its first pass the assembler processes the deck serially; that is, instructions are processed in the order in which they occur in the card deck, regardless of whether location counters or ORG's have been used and regardless of location counter hierarchy. When a symbol appears in the location field (i.e. it is defined) of any instruction (except SET), it is assigned a set-value of 1. Any symbol which is used before it is defined has a set-value of zero associated with it at that place in the deck. In the following example, at the point (A) the set-value of X is zero and the set-value of Y is 1, and at the point (B) the set value of both X and Y is 1:

```
          USE    STOR
Y         DEC    10
          USE
          CLA    X
          ADD    Y        (A)
           .
           .
           .
          USE    PREVIOUS
X         DEC    5
          USE    PREVIOUS
          DVH    X        (B)
```

The set-value of a symbol may be changed from 1 or zero to other values by using the symbol in a SET operation.

The SET Pseudo-Operation

The symbol appearing in the location field of a SET pseudo op is defined, or, if it has appeared in the location field of a previous SET, its set-value is redefined. Its new set-value is equal to the resulting set-value of the expression appearing in the variable field. The maximum set-value is 32767; set-values are modulo 32768. For example, consider the following sequence of instructions:

				s-value of J	s-value of K
	J	SET	10	10	0
(A)		AXT	K,4	10	0
	K	SET	21	10	21
	J	SET	K/J	2	21
	K	SET	J	2	2

Note that the instruction at (A) is assembled as AXT 0,4.

The primary utility of SET in macros is counting within an IRP loop.

Suppose in a program deck the following sequence of instructions appeared many times:

```
TRA     *+
PZE     A
PZE     B
 .
 .
 .
PZE     N
```

but each time the number of PZE's was different and each time the desired transfer was to the instruction immediately following the last PZE. Using the SET pseudo-op in an IRP loop enables a macro to be written to accomplish this task:

```
LOC2    MACRO   A
Z.Z.    SET     0
        IRP     A
Z.Z.    SET     Z.Z.+1
        IRP
        TRA     *+Z.Z.+1
        IRP     A
        PZE     A
        IRP
        ENDM    LOC2
```

To use LOC2:

LOC2 (A1,B1,C1,D1,E1)

The set-value of the symbol Z.Z. is used here as a counter. The first thing that is done in the macro definition is that it is SET to zero (or "initialized"). The first IRP loop is on "dummy" argument A, and Z.Z. is incremented by one each time through the loop. Since, as we have used it, there are five sub-argument (A1,B1,C1,D1, and E1) which replace the "dummy" argument A, at the conclusion of the last (fifth) trip through the first IRP the set-value of Z.Z. is 5. Since for this usage there will also be five PZE's (a PZE for each subargument) the desired transfer,*+Z.Z.+1, will be to *+6. Thus, the code generated will be:

```
TRA    *+6
PZE    A1
PZE    B1
PZE    C1
PZE    D1
PZE    E1
```

It can be seen that the macro LOC2 will be valid for any number of subarguments (PZE's desired).

It is the author's convention to write "Set symbols" of the form "Z.Z." only. The interspersed periods help to avoid inadvertent use of the "Set symbol" in the location field of any other operation than SET, which would result in the symbol being "improperly qualified".

IFT AND IFF, the "If True" and "If False" Pseudo-Operations

Most of the improvement in power in the IBMAP macro generator is due to the many new features of IFT and IFF. The "if" statement, used alone or in conjunction with other "if" statements, determines if the single next instruction (be it a hardware instruction, pseudo op, or macro instruction) immediately following the "if" statement (or group of "if" statements) will be assembled or not. If the condition specified by the "if" statement is met, or if the requisite conditions specified by a group of "if" statements acting in conjunction is met, the next single instruction is assembled; if the requisite condition(s) are not met, the next instruction is not assembled.

Elements may be compared in the variable field on an "if" statement by their set-values or by their BCD values. They may be compared on the basis of greater than, equal to, or less than, and successive "if's" may be combined by using the logical OR or the logical AND. Some examples:

```
ADDD        MACRO    A,B,C
            CLA      A
            ADD      B
            IFF      /C/=/STOR/
            STO      C
            ENDM     ADDD
```

If the parameter substituted for "dummy" argument C when the macro instruction ADDD is used is not literally the symbol "STOR", then and only then is the STO instruction assembled. Two uses of ADDD:

```
ADDD        X,Y,Z    ADDD        X,Y,STOR
```

produce:

```
CLA         X        CLA         X
ADD         Y        ADD         Y
STO         Z
```

The way to think of this is that STO Z is assembled because the condition of the "if" statement preceding it <u>was met</u>, i.e. it <u>is</u> false that "Z" is literally "STOR" - therefore assemble STO Z.

Suppose it is desired that the STO instruction be assembled only if the argument substituted for the "dummy" argument C is not STOR, and if the argument is a symbol that has been previously defined (we assume here that it has not appeared in a SET operation); i.e., the conditions for assembly of STO C are: "C"≠"STOR" and the set-value of "C"=1.

```
ADDD        MACRO    A,B,C
            CLA      A
            IFF      /C/=STOR/,AND
            IFT      C=1
            STO      C
            ENDM     ADDD
```

STO C will be assembled if and only if both conditions (because of the "AND") are met.

To assemble STO C if either condition (or both) is met, replace "AND" with "OR".

To assemble STO C only if the set-value of the argument which replaces C has a set-value <u>greater</u> <u>than</u> 5 (for example):

```
.
.
.
IFT     C=+5
STO     C
ENDM    ADDD
```

To assemble STO C only if the set-value of the argument which replaces C has a set-value less than 10 (for example):

```
.
.
.
IFT     C=-10
STO     C
ENDM    ADDD
```

To assemble STO C if the set-value of the argument which replaces C has a set-value greater than 5 but less than 10:

```
.
.
.
IFT     C=+5,AND
IFT     C=-10
STO     C
ENDM    ADDD
```

To assemble STO C if the set-value of the argument which replaces C has a set-value greater than 5, less than 10, and if it is not literally the symbol "STOR", or if and only if it is the symbol "BUD":

```
          .
          .
          .
          IFT         C=+5,AND
          IFT         C=-10,AND
          IFT         /C/=/STOR/,OR
          IFT         /C/=/BUD/
          STO         MAC
          ENDM        ADDD
```

If this last mentioned ADDD is used as follows:

```
1)  BUD     SET     3                3)  SAM     SET     7
            .                                    .
            .                                    .
            .                                    .
            ADDD    X,Y,BUD                      ADDD    X,Y,SAM
            .                                    .
            .                                    .
            .                                    .

2)  STOR    SET     9                4)  BOB     SET     5
            .                                    .
            .                                    .
            .                                    .
            ADDD    X,Y,STOR                     ADDD    X,Y,BOB
```

the following code is generated:

```
1)  CLA     X       2)  CLA  X      3)  CLA  X       4)  CLA  X
    ADD     Y           ADD  Y          ADD  Y           ADD  Y
    STO     MAC         STO  MAC
```

The relational operators "greater than", "equal to", and "less than" all may be used with BCD fields. However, these fields are compared on a left-justified, scientific collating sequence, and some care is required in handling these. For example, the statement

```
IFT        /10/=+/3/
```

is <u>not</u> true, and the next instruction will not be assembled.

The "if" statement, unlike IRP, may be used anywhere in the program and not just in macro definitions; however, it yields its greatest utility in macro definitions. It (singly or in conjoined groups) only affects the assembly of the single operation following, but that operation may be a macro operation which can expand to any length.

Utilizing IBMAP Macro Generating Power

Previously we discussed writing a macro definition to evaluate a polynomial equation of any order, using the IRP pseudo-operation. It was used to generate the code for (((AX + B)X + C)X + D)X + E as follows:

```
POLY     A(B,C,D,E)X
```

This generated:

```
CLA      A
XCA
FMP      X
FAD      B
XCA
FMP      X
FAD      C
XCA
FMP      X
FAD      D
XCA
FMP      X
FAD      E
```

Using IRP, SET, and the "if" statements, it is possible to write a macro definition for POLY which is used as follows:

```
POLY     (A,B,C,D,E)X
```

and which assembles LDQ A instead of $\left\{\begin{array}{ll}\text{CLA} & \text{A.}\\ \text{XCA} & \end{array}\right\}$

```
POLY      MACRO     COEFF,VAR
Z.Z.      SET       0
          IRP       COEFF
          IFT       Z.Z. =0
          LDQ       COEFF
          IFT       Z.Z. = +1
          XCA
          IFF       Z.Z. = 0
          FMP       VAR
          IFF       Z.Z. = 0
          FAD       COEFF
Z.Z.      SET       Z.Z. + 1
          IRP
          ENDM      POLY
```

Used POLY (A,B)X, which expands AX + B, the code generated is:

```
          LDQ       A
          FMP       X
          FAD       B
```

Z.Z. is initially cleared (the set-value) to zero. The rest of the macro definition is an IRP loop on the dummy variable COEFF. This loop, in expanding the macro as used, will be negotiated twice, once for COEFF=A and once for COEFF=B. On the first trip through, with COEFF=A, Z.Z. is zero; thus LDQ A is assembled but not XCA nor FMP X nor FAD A. Next Z.Z. is incremented to 1 and the IRP loop is reentered for COEFF=B. Neither LDQ B nor XCA are assembled, but, since Z.Z. is not zero, FMP X and FAD B are assembled and the macro is fully expanded. If the polynomial is greater than first order so that the IRP loop is negotiated three or more times, on the third and succeeding times through the loop (with Z.Z. greater than 1) the XCA instruction is assembled.

<u>Recursive and Non-Recursive Nested Macros</u>

A recursive macro is one which is used as a nested macro instruction within its own definition. In the following definition of a macro named POLY2 (this time to evaluate a third order or lower polynomial) the nested macro CYCLE is used: CYCLE uses the nested macro CYCLE in its definition - thus, it is recursive.

```
POLY2       MACRO    VAR,COEFF1,COEFF2,COEFF3,COEFF4
            CLA      COEFF1
            CYCLE    VAR,COEFF2,COEFF3,COEFF4
            ENDM     POLY2,NOCRS
CYCLE       MACRO    VAR,COEFF2,COEFF3,COEFF4
            XCA
            FMP      VAR
            FAD      COEFF2
            IFF      /COEFF3/=//
            CYCLE    VAR,COEFF3,COEFF4
            ENDM     CYCLE,NOCRS
```

To expand AX^3+BX^2+CX+D $(=((AX+B)X+C)X+D)$ code

```
            POLY2    X,A,B,C,D
```

POLY2 assembles CLA A, then "calls" CYCLE sending along the variable name and all but the first coefficient. CYCLE assembles XCA, FMP X, and FAD B. Since COEFF3 is not null (COEFF3=C), CYCLE "calls" itself passing along the variable name and all but the first two coefficients. But now, the "dummy" argument COEFF2 is replaced by the "dummy" argument COEFF3; i.e. the nested macro is replaced by CYCLE X,C,D. XCA,FMP X, and FAD C are now assembled. Since D is not null, CYCLE again calls itself: CYCLE X,D. XCA, FMP X and FAD D are now assembled. But there are no more coefficients after D so that the "dummy" variable COEFF3 is replaced by "null" and the macro expansion by recursion ceases.

NOCRS, in the second variable field of the ENDM card in the macro definition of POLY (and also CYCLE), signals the macro generator when expanding the macro instructions during the first assembler pass not to create symbols for substitutable arguments which are not replaced by real (not "null") arguments of the macro instruction used; i.e. NOCRS tells the macro generator to treat arguments which are not supplied to these macro instructions as if they were specifically "null" - blank or zero as appropriate. Otherwise the macro generator would create symbols (if in this mode) of the form ..nnnn (such as ..0001, ..0002, etc.) for these arguments. Since, in CYCLE, COEFF3 is compared literally to blanks and expansion ends when CYCLE is a "null" argument, symbols which might be created (without NOCRS) in the recursion must be suppressed in order to terminate the expansion by this method. If they are not suppressed the macro CYCLE would call itself unendingly - the assembler would "hang up" in a loop in its first pass. This error, called "circularity of definition", is to be avoided at all times, and it is the programmer's responsibility to detect this situation - not the assembler's.

A method of evaluating

C1*V1+C2*V2+C3*V3+...+CN*VN

allotting a cell to each product is as follows:

```
LDQ     C1
FMP     V1
STO     STOR
LDQ     C2
FMP     V2
STO     STOR+1
     .
     .
     .
LDQ     CN
FMP     VN
FAD     STOR+N-2
     .
     .
     .
FAD     STOR+1
FAD     STOR
```

where STOR is the first location of a block of temporary storage cells. A single macro instruction can be defined to do this expansion:

```
APROD1   MACRO   T,A              define a two argument macro to add products
  X.X.   SET     0                initially set X.X. to zero
         IRP     A                enter IRP loop to count subarguments of A
  X.X.   SET     X.X.+1           increment X.X. by 1 for each subargument of A
         IRP                      at end of IRP loop, X.X.=no. of subarguments of A
  X.X.   SET     X.X./2           X.X. (which was even) is halved
  Z.Z.   SET     0                initially set Z.Z. to zero
         IRP     A                enter multiply and store IRP loop
  Z.Z.   SET     Z.Z.+1           increment Z.Z. by 1 for each subargument of A
  A.A.   SET     Z.Z./2           A.A. set to Z.Z./2 truncated
  B.B.   SET     Z.Z.-2*A.A.      B.B. is 1 for odd Z.Z., B.B. is zero for even Z.Z.
         IFT     B.B.=1           B.B. is 1 when subargument of A is Cn
         LDQ     A                and LDQ Cn is assembled
         IFT     B.B.=0           B.B. is zero when subargument of A is Vn
         FMP     A                and FMP Vn is assembled
         IFT     A.A.=-X.X., AND  if the CnVn pair is not the last pair, and
         IFT     B.B.=0           if subargument of A is Vn,
         STO     T+A.A.-1         then assemble a STO into temporary storage
         IRP                      end multiply and store IRP loop
         IRP     A                enter FAD IRP loop
  Z.Z.   SET     Z.Z.-1           decrement Z.Z. by 1
  A.A.   SET     Z.Z./2           A.A. set to Z.Z./2 truncated
  B.B.   SET     Z.Z.-2*A.A.      B.B. is 1 for odd Z.Z., and zero for even Z.Z.
         IFF     A.A.=0, AND      if A.A. is not zero, and
         IFT     B.B.=0           if subargument of A is Vn(Z.Z. is even),
         FAD     T+A.A.-1         then assemble FAD from temporary storage
         IRP                      end FAD IRP loop
         ENDM    APROD1           end macro
```

When used: APROD1 STOR(C1,V1,C2,V2,C3,V3) the code outlined previously is generated.

It is also possible to solve this problem by defining a macro, call it APROD2, which calls two recursive macros, PROD and FADD, as follows:

```
APROD2  MACRO   T,C1,V1,C2,V2,C3,V3  define macro for max.no.of
                                     arg. desired
 Z.Z.   SET     -1                   nth product stored in T+n-1 cell
        PROD    T,C1,V1,C2,V2,C3,V3  do all multiplies and temp. stores
        IFF     Z.Z.=0               if there is more than one product
        FADD    T                    call FADD to generate sum
        ENDM    APROD2,NOCRS         end macro and suppress created
                                     symbols

PROD    MACRO   T,C1,V1,C2,V2,C3,V3  define macro for max. no. of
                                     arg. desired
 Z.Z.   SET     Z.Z.+1               increment product counter,
                                     Z.Z., by 1
        LDQ     C1                   assemble LDQ for present C1
        FMP     V1                   and FMP for present V1
        IFF     /C2/=//              if there are more products to
                                     assemble
        STO     T+Z.Z.               store this product in temp cell
        IFF     /C2/=//              if there are more products to
                                     assemble
        PROD    T,C2,V2,C3,V3        call PROD to assemble the next
                                     one
        ENDM    PROD,NOCRS           end macro and suppress created
                                     symbols

FADD    MACRO   T                    define macro with one argument
 Z.Z.   SET     Z.Z.-1               decrement product counter by 1
        FAD     T+Z.Z.               FAD temp storage cell (reverse
                                     order)
        IFF     Z.Z.=0               if there is still a product
                                     unsummed
        FADD    T                    call FADD to sum it
        ENDM    FADD
```

When used:

APROD2 STOR, C1,V1,C2,V2,C3,V3

the code outlined previously is again generated.

The question arises: Which APROD macro definition is preferable? There are several factors to be considered. APROD1, as is, can handle an indefinite number of products, whereas the definition of APROD2 (and PROD) must be changed to include the new arguments whenever a new maximum number (greater than the number in the present definition; i.e. 3) of products is required. APROD1, being longer than the combined lengths of APROD2, PROD, and FADD, requires more cells in the macro skeleton table, but requires fewer entries in the macro parameter table. However, APROD1 requires that 12P "if" statements (P is the number of products), 14P+3 SET statements, and 3 "IRP loops" be processed, while APROD2 (including its nested recursive macros) requires only the processing of 3P "if's", 2P SET's and 2P - 1 nested macros - resulting in relatively shorter assembly time for the expansion of APROD2 as compared with APROD1 (about 16 - 25% of the time).

Comparing the first non-nested, non-recursive POLY macro (defined in the section "IRP - the Indefinite Repeat Pseudo-Operation", page 9) with POLY2, page 18 (because they expand to the same code), one finds that POLY requires the processing of no "if's", no SET's and 1 "IRP loop", while POLY2 requires the processing of C "if's" (where C is the number of coefficients in the polynomial), no SET's and C nested macros. Obviously, POLY should assemble considerably faster than POLY2.

In general, however, it may be stated that a macro definition utilizing nested recursive macros will probably expand faster (i.e. require less assembly time) than the equivalent non-recursive macro whenever action is required in the "IRP loop" of the non-recursive macro which is dependent upon particular subarguments of the IRP parameter. To differentiate between subarguments requires "if" statements, (and usually SET's) and these require additional processing. Recursive macros, however, do not have to differentiate between subarguments (in fact "IRP loops" are rarely used here), because each parameter is separately named in the argument list.

The primary virtue, then, of recursive macros in replacing IRP's, "if's", and SET's is speed - and the primary virtue of IRP's, etc., in non-recursive macros is that the number of subarguments of an argument may be "limitless" and independent of the macro definition.

Non-recursive nested macros may also be of considerable utility to the programmer who is faced with a task of out-of-the ordinary complexity. For example, in exploring the possibilities of more efficient compilation of arithmetic statements, the author developed a package ("Mactran") of 20 macros (nested 11 deep) to accomplish the task of compilation.

ADDITIONAL ITEMS

PMC - the Print Macro Cards Pseudo Operation

If the programmer desires to have the complete expansion, including "if" and SET statements and mnemonics for all assembled and unassembled instructions and pseudo ops in the order in which they are processed, he may use the PMC operation. Coding

```
PMC     ON
```

will yield the complete expansion (IRP's are not printed)

```
PMC     OFF
```

(the normal mode) suppresses all expansion of macros except for assembled macro instruction names and arguments. Coding PMC with any other variable field reverses the setting of the switch. The switch setting may be changed as many times and at as many places in the program deck as desired by the insertion of the proper PMC cards.

ETC Cards in Macro Definitions and Instructions

If a macro definition requires more "dummy" arguments than can be put on the MACRO card, the ETC card may be used (the maximum number of substitutable arguments is still limited to 63). For example:

```
SMF     MACRO     AL,TRANS,BUF,HERE
        ETC       STOR,TCM
          .
          .
          .
```

A comma may also follow HERE, but is not required.

There are two ways to use ETC cards when using macro instructions. The first is similar to the use of ETC in the macro definition (above):

```
        .
        .
        .
        SMF             X,Y,Z,
        ETC             A,B,C
        .
        .
        .
```

In this case, however, the comma following Z is required. If an ETC card is required while writing out the substitutable subarguments replacing a "dummy" argument, the subarguments must extend into column 72 and then be continued on a following ETC card - even if this means breaking up a symbolic name. For example,

```
        .
        .
        8.              12-16                    72
        SMF             ERROR,ABC(MM,X,Y,...,SYM
        ETC             BOL,T,U,V,X)A,B,C
        .
        .
        .
```

An Additional Note on the SET Pseudo Operation

If the macro defined as follows:

```
     EXAMP      MACRO      A,B
      Z.Z.      SET        A
                IFT        B=6, AND
                IFT        Z.Z.=0
                CLA        XX
                ENDM       EXAMP
```

is used: EXAMP 13.5,6 where 13.5 is a floating point number,

then CLA XX

is assembled.

The points here emphasized are that the set-value of a floating point constant is zero (it cannot be differentiated in this respect from a virtual symbol), and that the set-value of a fixed point constant is equal to the value of the fixed point constant-modulo 32,768.

Discussion of the Operation and Use of the Macro Generator

The "device" that is called the "macro generator" is nothing more than the part of the IBMAP assembler that is responsible for the checking and special encoding of macro definitions into the macro skeleton table and for the checking and expansion of macro instructions where they occur in the program deck by decoding the definition in the macro skeleton table back into BCD-like representation. IBMAP is a two pass (really 2 1/2) assembler, but the action taken by the macro generator is in the first pass only. Thus, code "generated" in a macro expansion and programmer generated code look essentially the same to the second pass of the assembler.

The encoded form of the macro definition in the macro skeleton table is binary coded decimal, with special control character use being made of the octal numbers 75, 76, and 77 (which do not represent any BCD character). These control characters may appear either alone or with another control character and their meaning varies accordingly. They are used both to preface and to suffix segments of the BCD code. For example, the suffix 7577 signals to the macro generator that the end of the macro has been reached. The scheme is similar in general to the scheme used in PREST.

The encoding of the macro definition and the decoding of the definition for the expansion of each macro instruction encountered is accomplished quite rapidly. The "IRP loops" are actually expanded during the decoding process itself; in fact, the "opening" IRP pseudo-op is replaced in the macro skeleton table by the code 7676N, where N is the position of the "dummy" argument in the argument list (01, 02, etc.) and the "closing" IRP pseudo-op is replaced by 7677. Thus, the reason no IRP operation appears in the programmer's assembly listings, even with PMC ON, is because the BCD representation of this pseudo-op does not exist after the first assembly pass.

The "if" pseudo-operations and the SET pseudo-operation are also evaluated in the assembler's first pass. That this must be so can be seen from the following:

```
Z.Z.      SET      6
          IFT      Z.Z.=6
          XXX      A,B,C
           .
           .
           .
```

where XXX is a previously defined macro instruction. In order to expand the macro XXX it must first be known if it is true that the S-value of Z.Z. is 6. Since macro expansions occur in the first pass, so then must "if" and SET be evaluated in the first pass.

Some mention has already been made of the trade-offs involved in chosing the "best" way to write a macro definition to do a particular job (see the section "Recursive and Non-Recursive Nested Macros"). Basically, there are three factors to be considered: The efficiency of the object code that the macro produces, the time required by the assembler to expand the macro whenever it is used, and the number of macro definitions desired for one program deck (each program deck, since it is assembled separately, must have its own macro definitions). A macro definition that produces the shortest object code to do the job usually is longer than one which produces code which is not as efficient. The longer the definition is, the more pseudo ops ("if's", SET's, and IRP's particularly) it will usually contain and the longer the time that is usually required to expand each macro instruction. On the other hand, the shorter the definition is, the more room for additional definitions there is.

Obviously then, the answer to how a particular macro definition can best be written will depend on the circumstances. The author's general rule-of-thumb is: write the macro definition that will assemble the best object code consistent with the number of macro definitions necessary for the program deck - and let the assembly time take care of itself.

The amount of space available for macro definitions varies because the macro skeleton table shares a block of core with the macro parameter table. Macro skeleton table overflow is likely to occur when the number of lines of coded macro definitions "approaches" 400. The error message received is "Macro Skeleton Table Overflow, No More Definitions Accepted". It is an error of level 4. Also possible is the "Macro Parameter Push Down Table Overflow" message of severity 5.

Appendix A: A Set of Fortran-like Macro Definitions ("Mactran")

As part of a recent study--an attempt to read certain Fortran source statements (with possibly slight revision) directly into the IBMAP assembler to be translated and assembled by a group of macros ("Mactran") instead of by a compiler--a set of macro definitions were developed which might be of interest to the applications programmer. These Mactran definitions enable the programmer to use the Fortran "Go To", "Computed Go To", and "Do" and "Continue" statements, and simplified forms of the "Read" and "Write" statements.

The "Go To" and the "Computed Go To" Statements

```
GO.TO    MACRO    A,B              "Go To" or "Computed Go To" macro
 Z.Z.    SET      0                initialize transfer point counter
         IRP      A                enter "IRP loop" to count transfer
                                   points
 Z.Z.    SET      Z.Z.+1           increment transfer pt. counter by 1
         IRP                       end count loop
         IFF      Z.Z.=1           if this is a "Computed Go To"
         LAC      B,4              load XR4 with 2's comp of index, B
         IFF      Z.Z.=1           if this is a "Computed Go To"
         TXL      ERROR,4,-Z.Z.-1  test legitimacy of index
         IFF      Z.Z.=1           if this is a "Computed Go To"
         TRA      *,4              trans to proper transfer using
                                   index
         IRP      A                enter transfer point loop
         TRA      A                assemble a transfer for each trans
                                   point
         IRP                       end transfer point loop
         ENDM     GO.TO            end macro
```

```
Coding: 8        12-16
        GO.TO    (X1,Y1,Z1)AA
```

assemble the following code:

```
        LAC      AA,4
        TXL      ERROR,4,-4
        TRA      *,4
        TRA      X1
        TRA      Y1
        TRA      Z1
```

where ERROR is the name of the user supplied error routine for an incorrect index value. If no checking of index value is desired, the TXL instruction and the IFF operation preceding it may both be removed from the definition.

Note that the code produced by this definition is essentially identical to the code compiled by Fortran IV for the "Computed Go To" statement. Note also that GO.TO destroys the contents of index register 4; its contents should be saved prior to coding GO.TO if required. The GO.TO definition can be altered to do this (not shown).

```
Coding  8       12-16
        GO.TO   BLAZES
```

produces the single instruction:

```
        TRA     BLAZES
```

and index register 4 is not destroyed.

Imbedded blanks are not allowed in the names of macro instructions- hence the "period" in "GO.TO".

The "Do" and "Continue" Statements

```
DO       MACRO    A,B,C,D,E          form is:  DO A B=C,D,E   (FIV)
D.O.     SET      D.O. + 1           increment count of nested DO's by 1
X.X.     SET      D.O./2             truncate D.O./2
T.T.     SET      1 + D.O. - 2 *X.X. T.T., the tag, is first 2, then 1 for the next
                                       deeper DO, then 2
         SXA      'A' + 2, T.T.      store the tag
         D.FINE   D                  AXC D,T.T. or LAC D, T.T.
         TXI      *+1, T.T., - 1     decrease contents of T.T. by 1
         SXD      'A' + 1, T.T.      use C(T.T.) as test in TXH in CONTIN
         D.FINE   E                  AXC E(1 if E null), T.T. or LAC E,T.T.
         SXD      A,T.T.             use C(T.T.) as increment in TXI in CONTIN
         IFF      /C/ = /1/,OR       if C is not literally 1, or
         IFF      /E/ = //, AND      if E is not null, and
         IFF      /C/ = /E/          if C is not literally the same as E
         D.FINE   C                  then assemble AXC C,T.T. or LAC C,T.T.
         STL      'A' + 1            store loc of *+1 in addr. of TXH in CONTIN
         ENDM     DO,NOCRS           no created symbols

CONTIN   MACRO                       required:  form is:  A  CONTIN  (FIV)
X.X.     SET      D.O./2             truncate D.O./2
T.T.     SET      1+D.O.-2*X.X.      T.T., the tag, is first 2, then 1 for the next
                                       deeper DO, then 2
         TXI      *+1,T.T.,**        increment the tag by E
         TXH      **,T.T.,**         if C(T.T.)> D go to top of DO; otherwise NSI
         AXT      **,T.T.            restore the original tag
D.O.     SET      D.O.-1             decrease count of nested DO's by 1
         ENDM     CONTIN             end macro
D.FINE   MACRO    A                  generates AXC A or LAC A (FIV)
Z.Z.     SET      A                  Z.Z. is equal to the set-value of the argument
         IFT      Z.Z.= 0            if A is zero or null
Z.Z.     SET      1                  make Z.Z.=1
         IFT      /A/=/1/,OR         if A is 1, or
         IFT      /A/=//,OR          if A is null, or
```

```
IFT     Z.Z.=+1           if A is a number greater than 1
AXC     Z.Z.,T.T.         assemble an AXC
IFF     /A/=/1/, AND      if A is
IFF     /A/=//, AND       none of
IFF     Z.Z.=+1           the above
LAC     A,T.T.            assemble a LAC
ENDM    D.FINE,NOCRS      end macro
```

In Fortran one writes:

```
          DO 100       I=1,10,2 or
          DO 200       J=2,12        etc.
```

In IBMAP using the above Mactran macro definitions one may code

```
          8            12-16
          DO           AL,I,1,10,2     or
          DO           BB,J,2,12
              .
              .
              .
BB        CONTIN
              .
              .
              .
AL        CONTIN
```

Note that CONTIN must be coded at the end of the DO loop. Any symbolic name may be used for any argument (there are no fixed restrictions).

These Mactran DO loops may be nested to any level. The tag used in the first level is 2, the tag used in the second level is 1, the third 2, etc. The pertinent tag is automatically saved before entering each DO and restored upon leaving. The index is not stored at the top of the DO loop. If the programmer desires to store the updated index, the first instruction before the DO instruction should be "STZ index" and the first instruction following the DO instruction should be "SCA index, T.T."

In this form of DO and CONTIN the index is given an initial value of -C and is incremented by -E until it is greater than -D-1. This conforms to the Fortran IV convention for forward-stored arrays. To give the index an original value of C and to increment it by E until it surpasses D, conforming to the Fortran II convention, use the following Mactran definitions:

```
DO        MACRO    A,B,C,D,E          form is:  DO A B=C,D,E,       (FII)
D.O.      SET      D.O.+1             increment count of nested DO's by 1
X.X.      SET      D.O./2             truncate D.O./2
T.T.      SET      1+D.O.-2*X.X.      T.T.,the tag, is first 2, then 1 for the next
                                          deeper DO, then 2
          SXA      'A'+2,T.T.         store the tag
          D.FINE   D                  AXT D,T.T. or LXA D,T.T.
          SXD      'A'+1,T.T.         use C(T.T.) as test in TXL in CONTIN
          D.FINE   E                  AXT E(1 if E null), T.T. or LXA E, T.T.
          SXD      A,T.T.             use C(T.T.) as increment in TXI in CONTIN
          IFF      /C/=/1/,OR         if C is not literally 1, or
          IFF      /E/=//,AND         if E is not null, and
          IFF      /C/=/E/            if C is not literally the same as E
          D.FINE   C                  then assemble AXT C,T.T. or LXA C,T.T.
          STL      'A'+1              store loc. of *+1 in addr of TXL in CONTIN
          ENDM     DO,NOCRS           no created symbols

CONTIN    MACRO                       required:  form is: A CONTIN (FII)
X.X.      SET      D.O./2             truncate D.O./2
T.T.      SET      1+D.O.-2*X.X.      T.T., the tag, is first 2, then 1 for the next
                                        deeper DO, then 2
          TXI      *+1, T.T., **      increment the tag by E
          TXL      **,T.T.,**         if C(T.T.)≤D go to top of DO; otherwise NSI
          AXT      **,T.T.            restore the original tag
D.O.      SET      D.O.-1             decrease count of nested DO's by 1
          ENDM     CONTIN             end macro

D.FINE    MACRO    A                  generates AXT A or LXA A
Z.Z.      SET      A                  Z.Z. is equal to the set-value of the argument
          IFT      Z.Z.=0             if A is zero or null
Z.Z.      SET      1                  make Z.Z.=1
          IFT      /A/=/1/,OR         if A is 1, or
          IFT      /A/=//,OR          if A is null, or
          IFT      Z.Z.=+1            if A is a number greater than 1
          AXT      Z.Z.,T.T.          assemble an AXT
          IFF      /A/=/1/,AND        if A is
```

```
IFF      /A/=//, AND      none of
IFF      Z.Z.=+1          the above
LXA      A,T.T.           assemble an LXA
ENDM     D.FINE,NOCRS     end macro
```

Here, too, CONTIN must be coded at the end of the DO loop, any symbolic name may be used for any argument, the DO loops may be nested to any level, and the pertinent tag (2,1,2,...) is automatically saved before entering each DO and restored upon leaving the loop. The index is not stored at the top of the DO loop. If the programmer desires to store the updated index, the first instruction before the DO instruction should be "STZ index" and the first instruction following the DO instruction should be "SXA index, T.T."

If the following is coded:

```
     1              8             12-16
                    DO            FSAM,X,1,N   (Meaning: DO FSAM
                     .                          X=1,N)
                     .
                     .
     FSAM           CONTIN
```

the following is generated by each set of macros:

```
              FIV                               FII

SXA                    FSAM+2,2        SXA          FSAM+2,2
LAC                    N,2             LXA          N,2
TXI                    *+1,2,-1        SXD          FSAM+1,2
SXD                    FSAM+1,2        AXT          1,2
AXC                    1,2             SXD          FSAM,2
SXD                    FSAM,2          STL          FSAM+1
STL                    FSAM+1
            .                                   .
            .                                   .
            .                                   .
FSAM TXI               *+1,2**FSAM     TXI          *+1,2,**
TXH                    **,2,**         TXL          **,2,**
AXT                    **,2            AXT          **,2

                       DO GSAM,I,1,9,3        (meaning: DO GSAM I=1,9,3)
                           .
                           .
                           .
GSAM                   CONTIN
```

is assembled as follows - if it is the 6th nested DO loop in a "nest":

FIV				FII	
	SXA	GSAM+2,1		SXA	GSAM+2,1
	AXC	9,1		AXT	9,1
	TXI	*+1,1,-1		SXD	GSAM+1,1
	SXD	GSAM+1,1		AXT	3,1
	AXC	3,1		SXD	GSAM,1
	SXD	GSAM,1		AXT	1,1
	AXC	1,1		STL	GSAM+1
	STL	GSAM+1			
	.			.	
	.			.	
	.			.	
GSAM	TXI	*+1,1,**	GSAM	TXI	*+1,1,**
	TXH	**,1,**		TXL	**,1,**
	AXT	**,1		AXT	**,1

The "Read" and "Write" Statements

In Fortran IV one may write:

```
READ (5,100)A,B,C
                          and
WRITE (6,200)X,Y,Z,ZZ
```

Using the Mactran macro definitions one may write:

```
8           12-16
READ        (5,AL)A,B,C
                           and
WRITE       (6,SAM)X,Y,Z,ZZ
```

```
WRITE   MACRO   A,B,C,D,E,F        form: WRITE (A1,A2)B,C,D,E,F
        TSX     .FWRD.,4           TSX to Fortran IV library Write entry pt.
        S.M     (A)                assemble TXI and PZE's
        W.T     (B,C,D,E,F)        assemble CLA's and TSX's to .FCNV.,4
        TSX     .FFTL.,4           this ends the write using F IV routine
        ENDM    WRITE,NOCRS        no created symbols

READ    MACRO   A,B,C,D,E,F        form: READ (A1,A2)B,C,D,E,F
        TSX     .FRDD.,4           TSX to Fortran IV library read entry pt.
        S,M     (A)                assemble TXI and PZE's
        R.D     (B,C,D,E,F)        assemble TSX's to .FCNV.,4 and STO's
        TSX     .FRTN.,4           TSX to F IV routine entry for end of list
        ENDM    READ,NOCRS         no created symbols

S.M     MACRO   A                  assemble TXI and PZE's
        TXI     *+4,,2             assemble TXI around PZE's for 2 arguments
        PZE                        dummy PZE required for standard return
Z.Z.    SET     0                  initialize A pointer
        IRP     A                  enter PZE loop; will be traversed twice
Z.Z.    SET     Z.Z.+1             increment pointer by 1
        IFT     Z.Z.=1, AND        if  this subargument is A1, and
        IFT     A=-10              if A1 < 10
        PZE     .UNO'A'.           assemble PZE  .UNO'1 digit'
        IFT     Z.Z.=1,AND         if this subargument is A1, and
        IFF     A=-10              if A1 ≥10
        PZE     .UN'A'.            assemble PZE  .UN'2 digits'.
        IFT     Z.Z.=2             if this subargument is A2
        PZE     A                  assemble PZE with address of format
        IRP                        end of PZE loop
        ENDM    S.M                end of macro

W.T.    MACRO   A                  convert parameters for writing
        IRP     A                  enter convert loop
        IFF     /A/=//             if the subargument is not null
        CLA     A                  assemble CLA parameter to be written
        IFF     /A/=//             if the subargument is not null
        TSX     .FCNV.,4           assemble TSX to F IV convert routine
        IRP                        end convert loop
```

```
         ENDM     W.T              end macro

R.D      MACRO    A                convert parameters read in
         IRP      A                enter convert loop
         IFF      /A/=//           if the subargument is not null
         TSX      .FCNV.,4         assemble TSX to F IV convert routine
         IFF      /A/=//           if the subargument is not null
         STO      A                assemble STO for converted data
         IRP                       end convert loop
         ENDM     R.D              end macro
```

Thus, coding:

```
          READ          (5,AL)A,B,C
           .
           .
           .
          WRITE         (6,SAM)X,Y,Z,ZZ
```

produces the following code:

```
          TSX           .FRDD.,4
          TXI           *+4,,2
          PZE
          PZE           .UNO5.
          PZE           AL
          TSX           .FCNV.,4
          STO           A
          TSX           .FCNV.,4
          STO           B
          TSX           .FCNV.,4
          STO           C
          TSX           .FRTN.,4
           .
           .
           .
          TSX           .FWRD.,4
          TXI           *+4,,2
          PZE
          PZE           .UNO6.
          PZE           SAM
          CLA           X
          TSX           .FCNV.,4
          CLA           Y
          TSX           .FCNV.,4
          CLA           Z
          TSX           .FCNV.,4
          CLA           ZZ
          TSX           .FCNV.,4
          TSX           .FFIL.,4
```

This Mactran code is identical to the code produced by the Fortran IV compiler for equivalent source statements, except that Fortran compiles the pseudo-operation CALL instead of TSX, etc. As the READ and WRITE macros are defined here they handle up to 5 parameters in the I/O list. To increase the maximum number of parameters that they can handle the number of "dummy" arguments must be increased in the READ and WRITE macro definitions, and the number of subarguments of W.T and R.D must also be increased. (Note that these macros embody a technique for making the arguments of one macro subarguments of a lower nested macro.)

For example, to increase the list capability of WRITE to a maximum of seven parameters the following changes are required:

```
        WRITE        MACRO        A,B,C,D,E,F,G,H
```

and

```
                     W.T          (B,C,D,E,F,G,H)
```

Note that these definitions do <u>not</u> offer the capability of the "implied DO loop" in the argument list that Fortran does.

These Mactran definitions use the Fortran IV Read-Write Decimal library routine, FWRD, which in turn uses IOCS, so that all I/O using these instructions is fully and automatically overlapped.

The "format statement" referenced in both the READ and WRITE instructions must conform to Fortran standards. It must be in BCD form, must begin and end with opening and closing parentheses, and must contain only matched pairs of parentheses (except in a Hollerith field). For example, WRITE (6,BBB) refers to the format statement at BBB. At BBB one might find:

```
BBB    BCI     6,(1H023X21HSTD.     FORT.  CONVENTION)
```

Again READ (5,XXX)A,B,C refers to the format statement XXX. At XXX one might find:

```
XXX   BCI      3,(F15.4,I2/E12.5)
```

Blanks in "format statements" are permissable. See the Systems Reference Library manual "IBM 7090/7094 Programming Systems, Fortran IV Language" C28-6274, for detailed information on READ,WRITE, and FORMAT statements.

Appendix B: Bibliography and References

1. "IBM 7090/7094 Programming Systems, MAP(Macro Assembly Program) Language", Systems Reference Library, C28-6311-2, Programming Systems Publications, Poughkeepsie, New York February, 1964.

2. "IBM 7090/7094 Programming Systems, Fortran IV Language", Systems Reference Library, C28-6274-1, Programming Systems Publications, Poughkeepsie, New York, May, 1963.

3. "Preliminary Systems Guide for IBM 7090/7094 Macro Assembly Program (IBMAP)", IBM 704/709/7090 Program Library, 7090-SP-804, D. S. Programming Systems, 1 September 1963.

4. "Mactran", by Gerald M. Berns unpublished, November, 1963